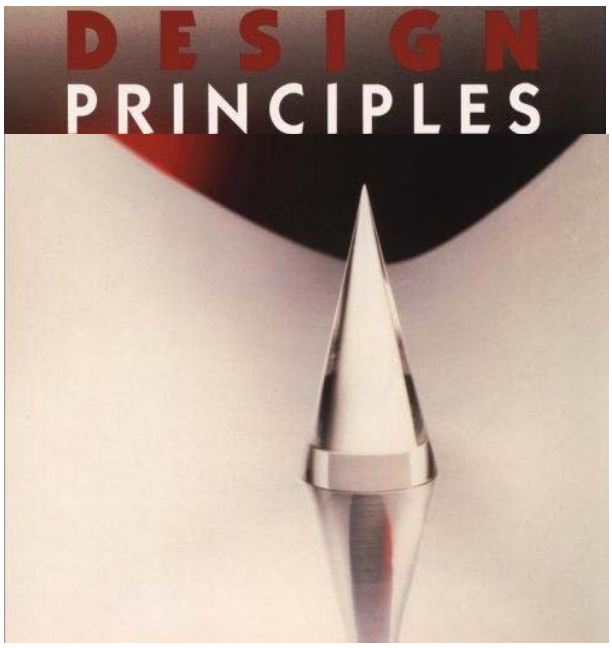
Object-Oriented Analysis and Design using UML and Patterns

Design Smells &
Advanced Design Principles



Speculative Design vs. Agile Design

Speculative Design

Anticipate changes to the requirements, and put facilities in design to cope with those potential changes.

Agile Design

Continuously improve the design so that it is as good as it can be for the system as it is *now for the current iteration*.

The most culprit for the bad design is the *interdependence* among modules

Dependence Management (DM) is an issue of loose coupling and strong cohesion.

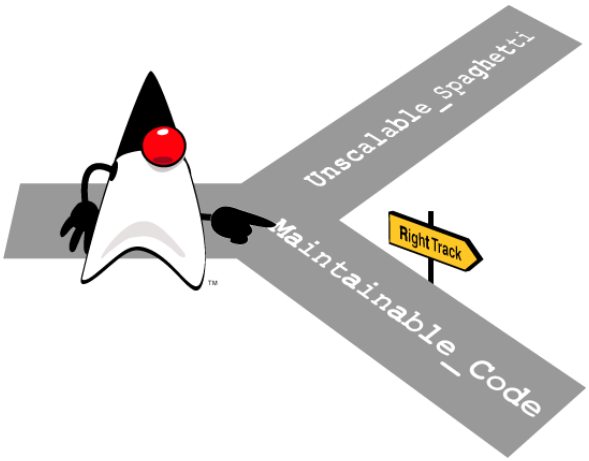
Controlling dependencies has several advantages for software system
- Not rigid, not fragile, reusable, low viscosity

Also affects development environment
- Lower compile and link times, less testing
- More productive work

What are the advantages for practicing good DM?

What is the penalties for practicing poor DM?

A system with poor dependency structure will typically exhibit four negative traits



Software system is *rigid*

Software system is *fragile*

Software system is *not reusable*

Software system has *high viscosity*

Rigidity is a tendency of software to be difficult to change, even in simple ways

A single change can cause a cascade of subsequent changes in dependent modules.

The impact of a change cannot be predicted and estimated.

Managers become reluctant to allow changes
- *"Don't touch a working system!"*

Fragility is a tendency of software to break in many places when a single change is made

A small change has large, non-local side effects.

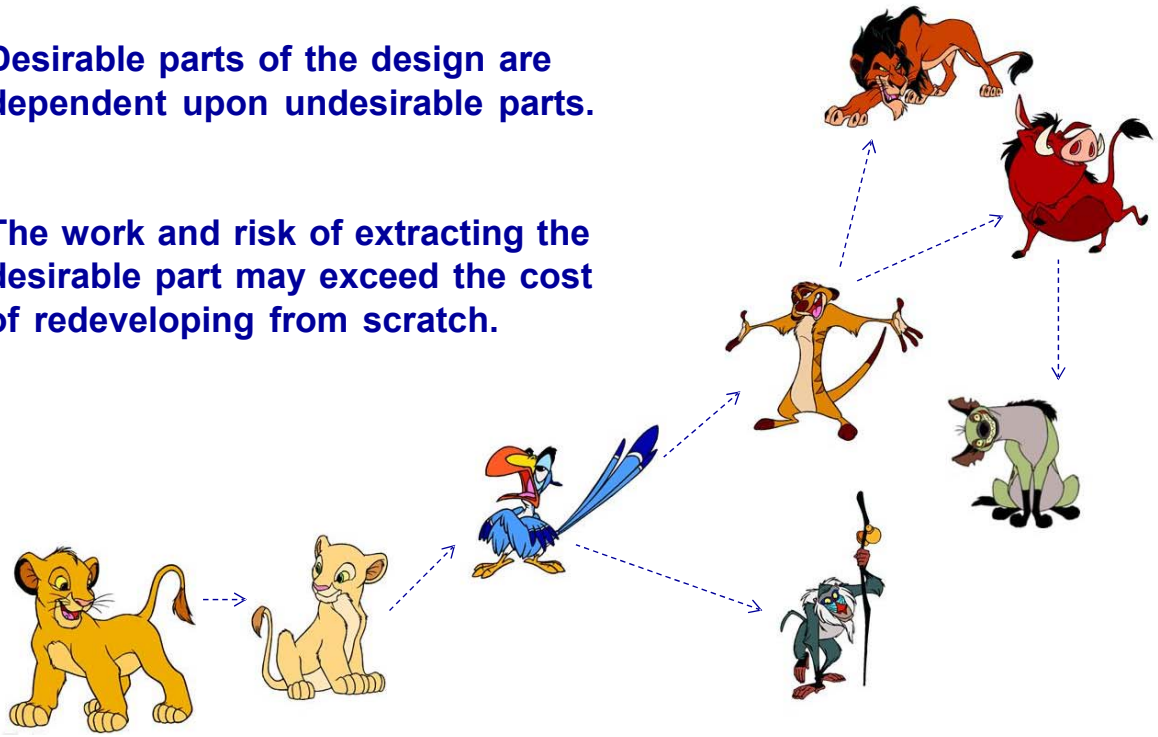
New errors appear in areas that seem unrelated to the changed areas.

It looks like control has been lost
- Users become critical
- Program looses credibility
- Developers loose credibility

Software is not reusable if it is so tangled that it's impossible to reuse anything

Desirable parts of the design are dependent upon undesirable parts.

The work and risk of extracting the desirable part may exceed the cost of redeveloping from scratch.



Viscosity: doing things right is harder than doing things wrong

Viscosity of Software

Hard to make changes without breaking original design

- make hacks, instead

Viscosity of Environment

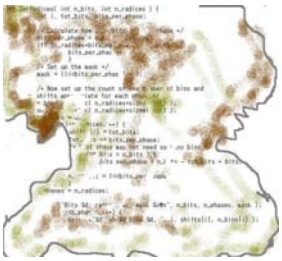
Slow and inefficient development environment

- Large tendency of programmers to keep changes localised even if they break designs

What stimulates the “Software to Rot”?

Requirements are the most volatile elements in the project.

Designs degrade because requirements change in ways that the initial design did not anticipated.



A case study “The Copy Routine”

Design Smells – The Odors of Rotting Software

Rigidity

Fragility

Immobility

Viscosity

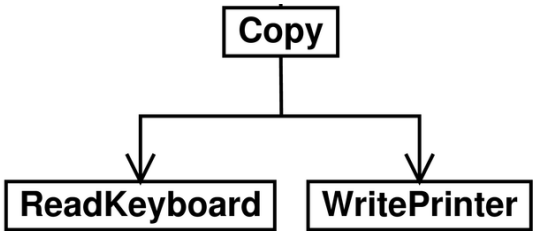
Needless Complexity

Needless Repetition

Opacity

Version #1

If lucky, all designs start well!




```
void copy(void)
{
     int ch;
     while( (ch=ReadKeyboard()) != EOF)
          WritePrinter(ch);
}
```

A simple and elegant solution to a simple problem

ReadKeyboard and WritePrinter may be reusable!

Version #2

We sometimes want to read from paper tape reader, too.

Hundreds of users already are using version #1 already.

```
bool GtapeReader = false; // remember to clear

void copy(void)
{
    int ch;
    while( (ch=GtapeReader ? ReadTape() : ReadKeyboard()) != EOF)
       WritePrinter(ch);
}
```

Ok! Just put in a global flag ➔ make sure to clear the flag after

It is backward compatible!

Version #3

Sometimes we need to write to a paper tape punch, still want to be backward compatible.

We've had this problem before, and just add another flag.

```
bool GtapeReader = false;
Bool GtapePunch = false;
// remember to clear

void copy(void)
{
    int ch;
    while( (ch=GtapeReader ? ReadTape() : ReadKeyboard()) != EOF)
       GtapePunch ? WritePunch(ch) : WritePrinter(ch);
}
```

The Copy routine seems to grow in size and complexity every time a new feature is added.

The protocol to use it becomes more complicated.

Examples of a Good Design in C!

```
#include <stdio.h>

void Copy( FILE* in, FILE* out ) {
  int ch;
  while( (ch= fgetc( in )) != EOF ) {
    fputc( ch, out );
  }
}
```

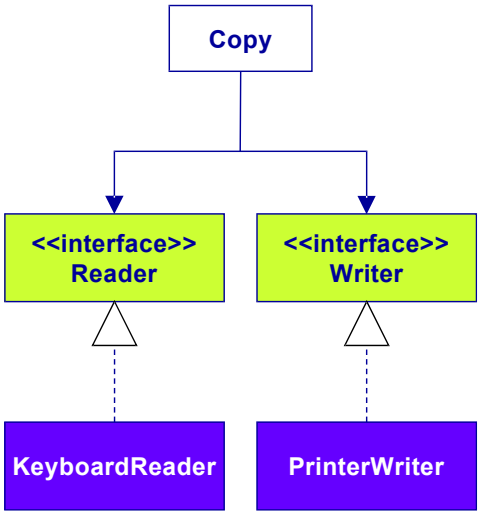
This program is based on abstraction, that is a "`FILE`"

The `FILE` looks like an interface.

The `fgetc` and `fputc` are dynamically bound.

But, nevertheless it is C!!

Object-Oriented Solution




```
interface Reader
{ char read(); }

interface Writer
{ void write(char c); }

public class Copy
{
    Copy(Reader r, Writer w)
    {
        itsReader = r;
        itsWriter = w;
    }
    public void copy()
    {
        int c;
        while( (c==itsReader.read()) != EOF )
            itsWriter.write(c);
    }
    private Reader itsReader;
    private Writer  itsWriter;
}
```


Advanced Principles of Object-Oriented Design

SRP: The Single Responsibility Principle

OCP: The Open Closed Principle

LSP: The Liskov Substitution Principle

ISP: The Interface Segregation Principle

DIP: The Dependency Inversion Principle

SRP: *A class should have one, and only one reason to change*

A class should be well-named, should have minimal and complete operations.

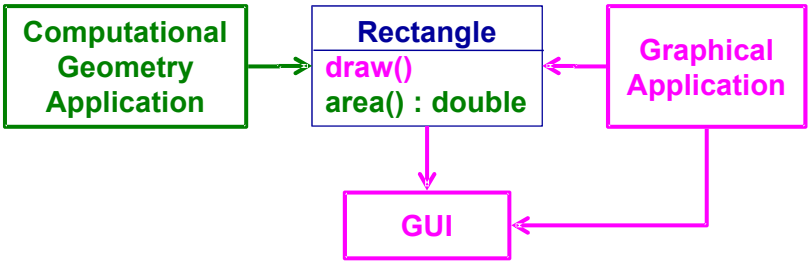
Operations in the class should be closely related to one subject area → *strong cohesion*

Shift the meaning a little bit, then operations in the class should be closely related to one responsibility.

In the context of the SRP, a responsibility is defined as *"a reason for change"* or *"axis of change"*

If you can think of more than one motive for changing a class, you are violating the SRP.

Violation of the SRP causes the smell of rigidity and/or fragility of the code

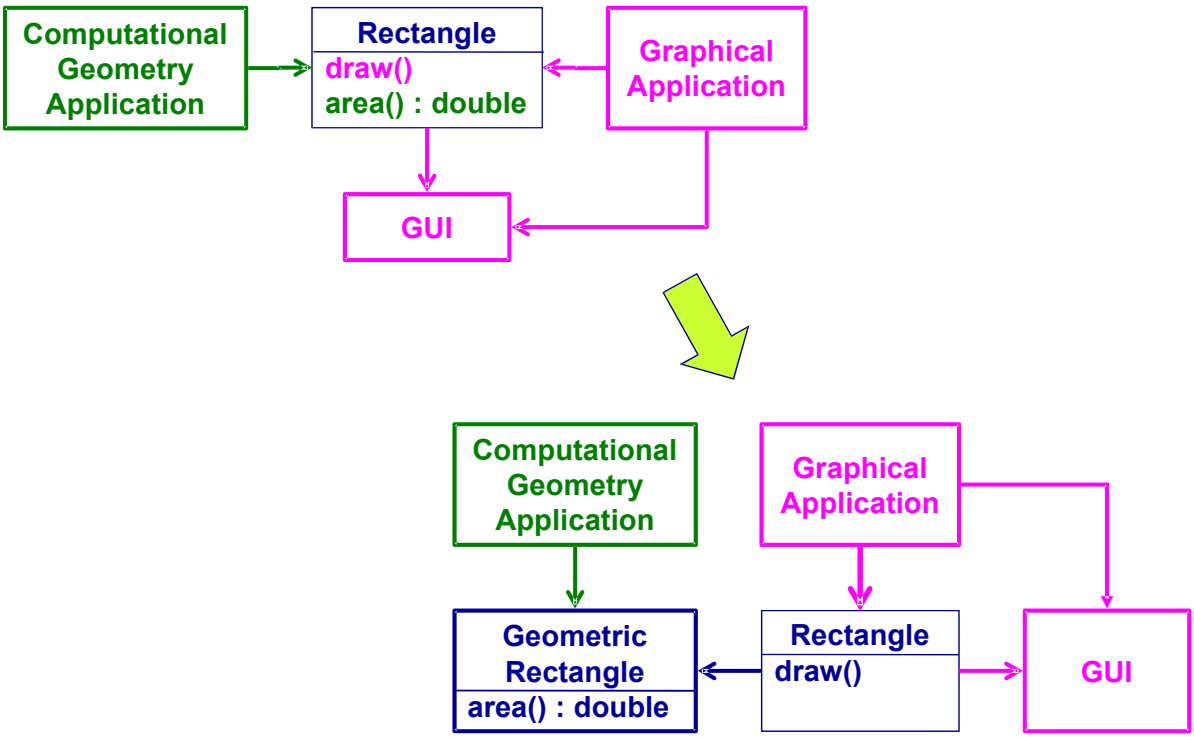


`Rectangle` has two responsibilities:
- one for rendering,
- another for calculation

CGA must include GUI for no good reason.

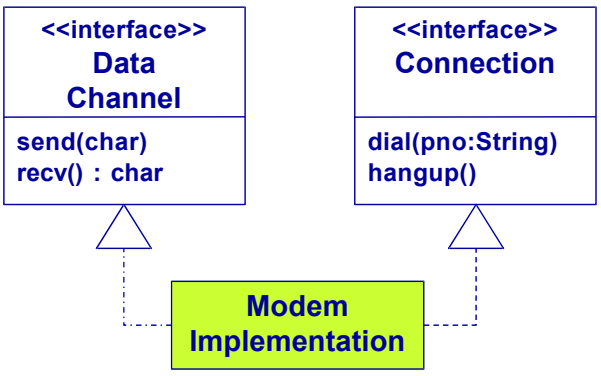
If a change in GA causes the `Rectangle` to change for some reason, that change may force us to rebuild, retest, and redeploy the CGA. Otherwise, CGA may break in unpredictable ways.

The SRP applied to the previous example



An axis of change is an axis of change only if the change actually occurs

```
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```



If application does not change in ways that cause the two responsibilities to change at different times, separating them would smell of *needless complexity*.

The SRP is one of the most simplest of the principles, and one of the hardest to get right.

Review Questions

What is the dependency management (DM)?

What are the symptoms of poorly managed dependency In software design?

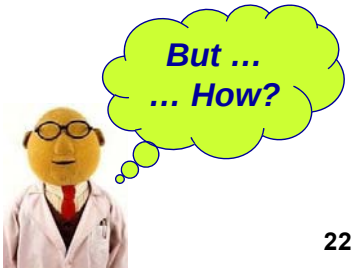
What is the SRP? Why is it important?

We must design modules that never change

All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version" --- *Ivar Jacobson*

When a change to a program results in a cascade of changes to dependent modules, that program is fragile, rigid, unpredictable and un-reusable.

Therefore, you should design modules that never change!



What are the design principles underlying the following heuristics?

Make all object-data private!

No global variables!

RTTI is ugly and dangerous!

OCP: The Open-Closed Principle

Software entities (classes, modules, functions etc.) should be open for extension, but closed for modification

- Bertrand Meyer, 1988 -

The OCP is the single most important guide for the OO designers.

Modules that conform to the OCP have two primary attributes

1. **Open for Extension**
   The behavior of the module can be extended.

2. **Closed for Modification**
   The source code of a module should not be changed.

How to resolve these two conflicting attributes?

If you want to extend the behavior of a module, do it by *adding new code* rather than by changing working code.

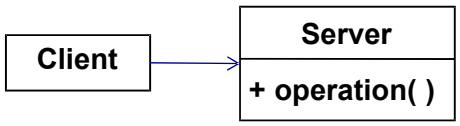
Create abstractions that are fixed and yet represent an unbounded group of possible behaviors

The abstractions are *abstract base classes*, and the unbounded group of possible behaviors is represented by all the possible derivative classes.

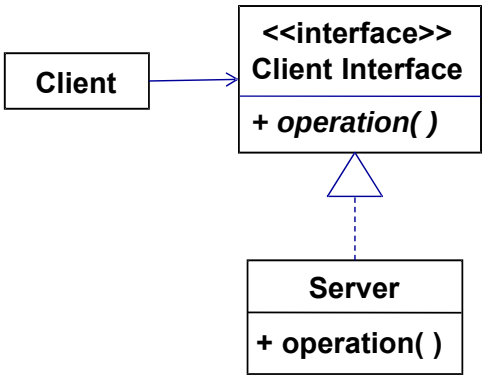
A module that manipulates only an abstraction can be closed for modification since it depends on an abstraction that is fixed.

Yet, the behavior of the module can be extended by creating new derivatives of the abstraction.

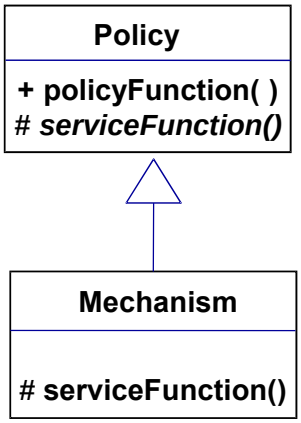
Two common ways to satisfy the OCP



Neither open nor closed



Both open and closed



Base class is open and closed (Template Method Pattern)

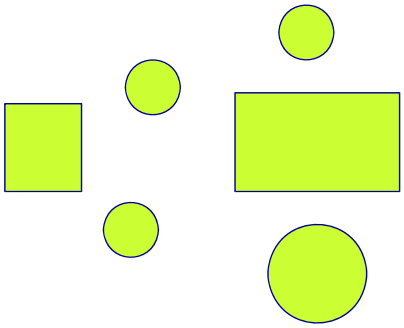
The key is a clear separation of generic functionality from the detailed implementation of that functionality.

Example: Problem Statement

We have an application that must be able to draw circles and squares on a standard GUI.

The circles and squares must be drawn in a particular order.

A list of the circles and squares will be in the appropriate order and the programming must walk the list in that order and draw each circle or square.



Procedural Solution Violating OCP

```
enum ShapeType {Circle, Square};
struct Shape { ShapeType type; };
struct Circle { ShapeType type; double radius; Point center; };
struct Square { ShapeType type; double side; Point origin; };
void drawCircle(struct Square*);
void drawSquare(struct Circle*);
typedef struct Shape* ShapePointer;
void drawAllShapes(ShapePointer list[], int n) {
  for (int i = 0; i < n; i++) {
    ShapePointer* s = list[i];
    switch (s->type) {
    case Square:
      drawSquare(static_cast<struct Square*>(s)); break;
    case Circle:
      drawCircle(staticic_cast<struct Circle*>(s)); break;
    }
}
```

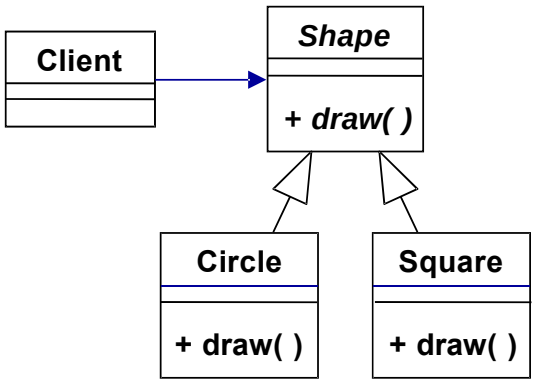

OO Solution Violating OCP

```
enum ShapeType {Circle, Square};
class Shape { public: ShapeType type; ~Shape() = 0; };
class Circle : public Shape { public: void drawCircle(); ... };
class Square : public Shape { public: void drawSquare(); ... };

void drawAllShapes(vector<Shape*>& list) {
  for (vector<Shape*>::const_iterator i = list.begin();
       i != list.end(); i++) {
    switch ((*i)->type) {
    case Square:
      dynamic_cast<Square*>(*i)->drawSquare();
      break;
    case Circle:
      dynamic_cast<Circle*>(*i)->drawCircle();
      break;
    }
}
```

OO Solution Conforming to OCP

```
class Shape {
public:
  virtual void draw() const = 0;
};
class Square : public Shape {
public:
  virtual void draw() const;
};
class Circle : public Shape {
public:
  virtual void draw() const;
};
void drawAllShapes(vector<Shape*>& list)
{
  for (vector<Shape*>::const_iterator
       i = list.begin();
       i != list.end()
       i++)
    (*i)->draw();
}
```



Since closure cannot be complete, it must be strategic

No significant program can be 100% closed!

The designer must choose the kind of changes against which to close his design.

The designer must guess the most likely kinds of changes, and then construct abstractions to protect him from those changes

- *ex) draw Squares before Circles*

A Word of Caution: Resisting premature abstraction is as important as abstraction itself!

- *Apply abstraction only to those parts of program that exhibit frequent changes*

The primary mechanisms behind the OCP are abstraction and polymorphism

One of the primary mechanisms that supports abstraction and polymorphism is inheritance

What are the design rules that govern the use of inheritance to support the OCP?

What are the characteristics of the best inheritance hierarchy?

What are the traps that will cause us to create hierarchies that do not conform to the OCP?

LSP: The Liskov Substitution Principle

Derived classes must be usable through the base class interface without the need for the user to know the difference

- Barbara Liskov, 1988 -

Subtypes must be substitutable for their base types (super types).

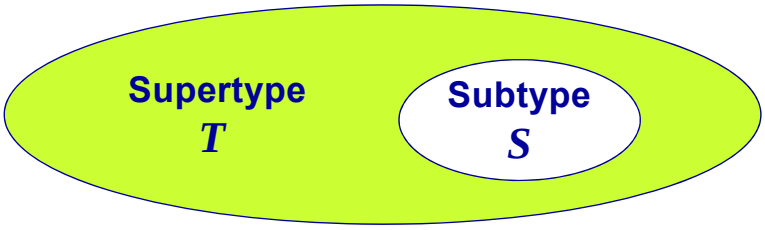
Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

Original Definition of Subtypes

What is wanted here something like the following property:
If for each object o_1 ***of type S there is an object*** o_2 ***of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when*** o_1 ***is substituted for*** o_2 ***then S is a subtype of T.***

In set theory,

$$Set\ (objects\ of\ type\ S) \subseteq Set\ (objects\ of\ type\ T)$$



Violation of the LSP: Simple Example

```
class Shape { public: virtual ~Shape() = 0; };
class Circle : public Shape { public: void drawCircle(); ... };
class Square : public Shape { public: void drawSquare(); ... };

void drawShape(const Shape& s)
{
    if (typeid(s) == typeid(Square))
        dynamic_cast<Square&>(s).drawSquare();
    else if (typeid(s) == typeid(Circle))
        dynamic_cast<Square&>(s).drawCircle();
}
```

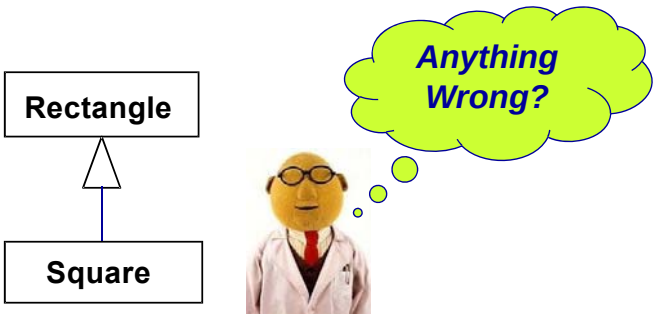
Misuse of C++ Run-Time Type Information (RTTI) to select a function based on the type of an object.

drawShape must be changed whenever new derivatives of the Shape class are added → violation of the OCP

Note: Violation of the LSP is a latent violation of OCP!

Square and Rectangle, a More Subtle Violation

A square *is a* rectangle for all normal intents and purposes in general. Since *"is a"* relationship holds, it seems logical to model the Square class as being derived from Rectangle.




```
class Rectangle {
public:
  void   setWidth(double w)
         { width = w; }
  void   setHeight(double h)
         { height = h; }
  double getHeight() const
         { return height; }
  double getWidth() const
         { return width; }
private:
  double width;
  double height;
};
class Square :
      public Rectangle { ... };
```


First, memory is wasted for squares → Let's ignore this …

Second, Square inherits setWidth and setHeight which are utterly inappropriate for a Square, since the width and height of a square is identical!

Solution?

```
void Square::setWidth(double w) {
  Rectangle::setWidth(w);
  Rectangle::setHeight(w);
}
void Square::setHeight(double h) {
  Rectangle::setWidth(w);
  Rectangle::setHeight(w);
}

Square s;
s.setWidth(1);  // Fortunately set the height to 1 too
s.setHeight(2); // sets width and height to 2, good thing
```

Now, the invariant of the square is maintained even if someone changes its height or width. *Are you happy?*

Let's consider the following:

```
void func(Rectangle& r) {
  r.setWidth(32);      // calls Rectanlge::setWidth(), Why?
}
```

If we pass a *reference-to*-Square object into this function, the Square object will be corrupted because the height won't be changed. This is a clear violation of LSP. The func does not work for derivatives of its arguments.

Solution? *Ah Hah! Make them virtual!*

By now, we might conclude that the model is now self consistent and correct. But wait! Consider the following:

```
void func(Rectangle& r) {
  r.setWidth(5);
  r.setHeight(4);
  assert(r.getWidth()*r.getHeight() == 20);
}
```

The programmer of func made a very reasonable assumption that *changing the width of a Rectangle leaves its height unchanged*. However, passing a reference to Square object to this function break this assumption.

This function exposes a violation of the LSP: the addition of the Square derivative of Rectangle has broken this function; and so the OCP has been violated.

What Went Wrong (W^3)?

Why did the apparently reasonable model of the Square and Rectangle go bad? After all, isn't a Square a Rectangle? Doesn't the "is a" relationship hold?

The answer is *NO! Why?*

Because the *behavior* of a Square object is not consistent with that of a Rectangle object. Behaviorally, a Square is not a Rectangle! And it is the *behavior* that software is all about.

The validity of a model can only be expressed in terms of its clients – Validity is not Intrinsic

A model, viewed in isolation, cannot be meaningfully validated

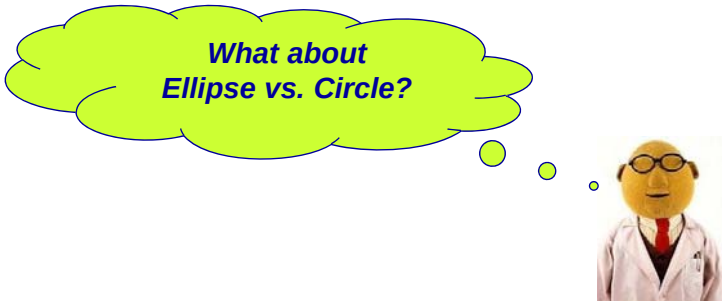
For example, the final versions of Square and Rectangle models were self-consistent and valid when considered in isolation. But when we look at them from a view point of a client who made a reasonable assumptions about the base class, the model broke down.

Thus, when considering whether a particular design is appropriate or not, one must *view the model in terms of the reasonable assumptions that will be made by the clients of that design*

In order for the LSP to hold, all derivatives must conform to the extrinsic behavior that clients expect of the base classes

The LSP makes clear that in OOD, the "is a" relationship pertains to *extrinsic public behavior*; behavior that clients depend on.

For example, clients of the Rectangle assume that the width and height are independent variables, and this independence is the extrinsic behavior that other clients are likely to depend on.



There is a strong relationship between the LSP and the concept of Design by Contract

A contract is an advertised behavior of an object suggest by Bertrand Meyer (1988).

Associated with methods of classes are *preconditions* and *postconditions:* *{Precondition} op {Postcondition}*

The precondition must be true in order for the method to execute. Upon completion, the method guarantees that the postcondition will be true.

- **precondition denotes *requirements***
- **postcondition denotes *promises***

For example, postcondition of Rectangle::setWidth(double w) is

```
assert((width==w) && (height==old.height));
```

"…when redefining a routine [in a derivative], you may only replace its precondition by an equal or weaker ***one, and its postcondition by a equal or*** stronger ***one."***

A Derived class should *require no more* and *promise no less*.

```
int Base::f(int x);
// REQUIRE: x is odd
// PROMISE: return int
```

```
int Derived::f(int x);
// REQUIRE: x is int
// PROMISE: return positive int
```

By the same argument, derived class should not throw exceptions whose base class don't throw!

DIP: The Dependency Inversion Principle

High level modules should not depend upon low level modules. Both should depend on abstractions.
Abstractions should not depend upon details.
Details should depend upon abstraction.

OCP states the goal; DIP states the mechanism.

A base class in an inheritance hierarchy should not know any of its subclasses.

Modules with detailed implementations are not depended upon, but depend themselves upon abstractions.

We want to reuse high level policies

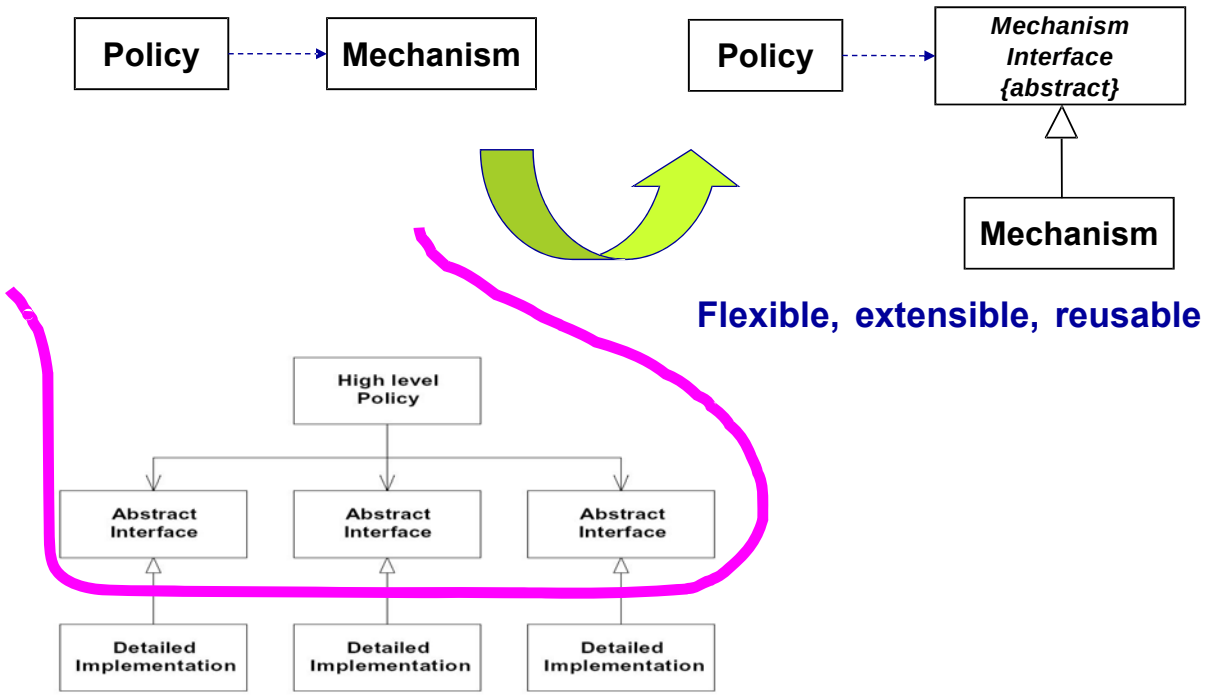
The high level modules contain the important policy decisions and business models of an application.

When high level modules depend upon low level implementation modules, it becomes very difficult to reuse those high level modules in different contexts.

However, when the high level modules are independent of the low level modules, then the high level modules can be reused quite simply.

This is the heart of the framework design.

Separate Policy from Mechanism (Layering)

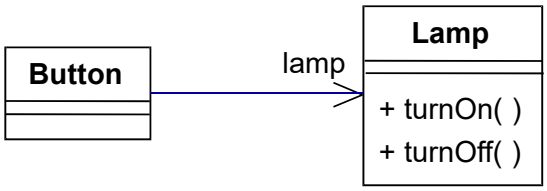


Example: Button/Lamp

```
//------ Lamp.h ----------
class Lamp {
public:
  void turnOn();
  void turnOff();
};
```

```
//------ Button.C --------
#include "Button.h"
#include "Lamp.h"
void Button::detect() {
  bool buttonOn =
       getPhysicalState();
  if (buttonOn)
    lamp->turnOn();
  else
    lamp->turnOff();
}
```

```
//---- Button.h ----
class Lamp;
class Button {
public:
  Button(Lamp& l)
  :lamp(&l){}
  void detect();
private:
  Lamp* lamp;
};
```



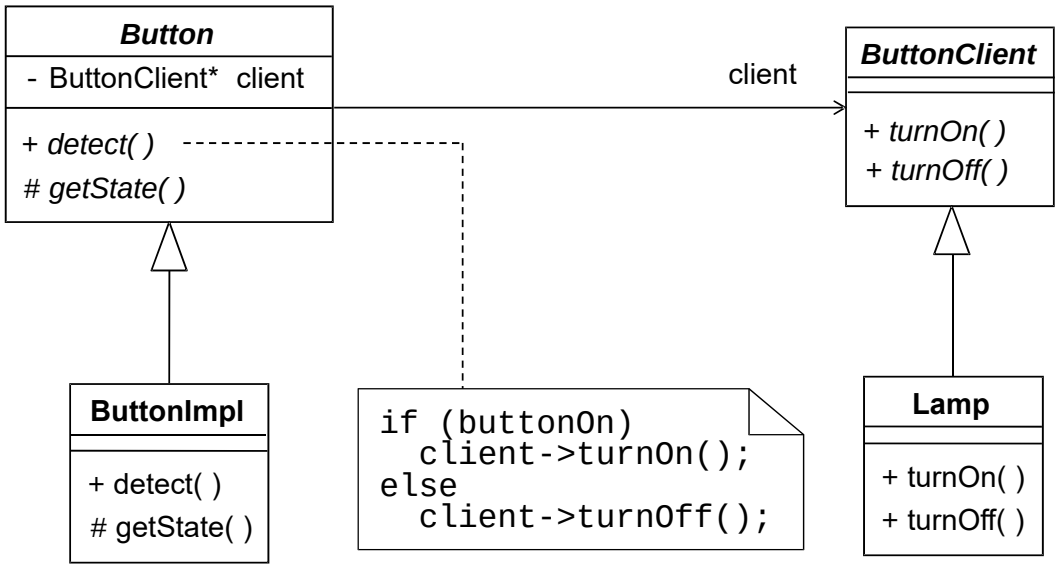
To conform to the DIP, we must isolate high level policy from the details of the problem

The high level policy is the abstractions that underlie the application, the truths that do not vary when the details are changed.

Abstraction → *To detect an on/off gesture from a user and relay that gesture to a target object*

Now, we must direct the dependencies of the design such that the details depend upon the abstractions.

Inverted Button/Lamp



Related Heuristics to DIP

Program to an interface, not an implementation!

Avoid transitive dependencies!

When in doubt, add a level of indirection!
- **It is generally easier to remove or by-pass existing levels of indirection than it is to add them later**

ISP: The Interface Segregation Principle

> ***Clients should not be forced to depend upon interfaces that they do not use***

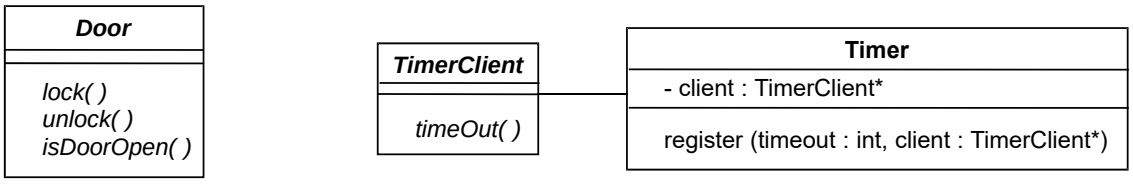
The ISP is a structural principle like DIP.

Deals with the disadvantages of "fat" interface

-- fat interface, polluted interface, non-cohesive interface

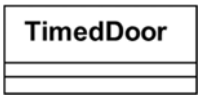
Violation of the ISP violates the SRP.

Interface Pollution Example



Your Own **Third Party**

For a security system, we need a TimedDoor object which sounds an alarm when the door has been left open for too long. In order to do this the TimedDoor is going to use Timer.



Need to act as both TimerClient and Door. But *how*?

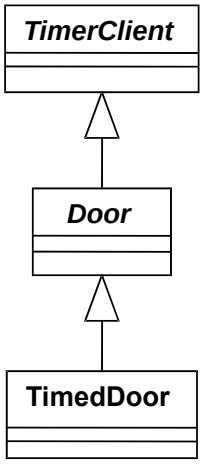
One Possible Solution, Problematic Solution

The Door class is now dependent on TimerClient.

Timing free derivatives of the Door will have to provide *nil* implementation for the timeOut method.

Moreover, the applications that use those derivatives will have to #include the definition of the TimerClient, even though it is not used.

The Door interface has been polluted with an interface that it does not require.



Separate Clients mean Separate Interfaces

Door and TimerClient represent interfaces that are used by completely different clients.

Since clients are separate, the interfaces should remain separate too. *Why?*

Because, as you will see soon, *clients exert forces upon their server interfaces.*

Risk of Interface Pollution

Normally, changes in the interface of a class affect the user of the class. However, sometimes it is the user that forces a change to the interface → *backward force*

What will happen if some users of Timer register more than one timeOut request?

False alarm may occur! → Design flaw in the first place

```
class Timer {
public:
  void register(int timeout, int timeOutId, TimerClient* client);
};
class TimerClient {
public:
  virtual void timeOut(int timeOutId) = 0;
};
```


This change will affect all the users of TimerClient. That is OK because we acknowledge our design mistake.

However, this change will also cause Door and all clients of Door to be affected (at least recompiled) by this fix.

Why should a bug in TimerClient have any affect on clients of Door derivatives that do not require timing?

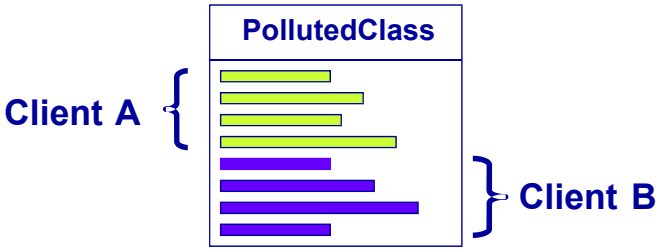
When a change in one part of the program affects other completely unrelated parts of the program, the cost and repercussions of the changes become unpredictable.

Lessons Learned

When clients are forced to depend upon interfaces that they don't use, then those clients are subject to changes to those interfaces.

Said another way, when a client depends on a class that contains interfaces that the client does not use, but that other clients *do* use, then that client will be affected by the changes that those other clients force upon the class.

Therefore, we would like to avoid such couplings where possible, and so we want to separate the interfaces where possible.



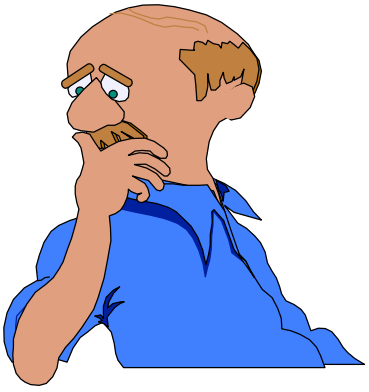
Dilemma .....

TimedDoor, we need to use two separate interfaces (TimerClient and Door) used by two separate clients.

These two interfaces *must* be implemented in one object:
➔ TimedDoor

So how can we conform to the ISP?

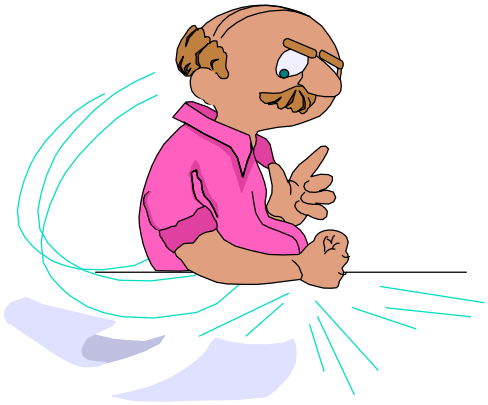
How can we separate the interfaces when they must remain together?



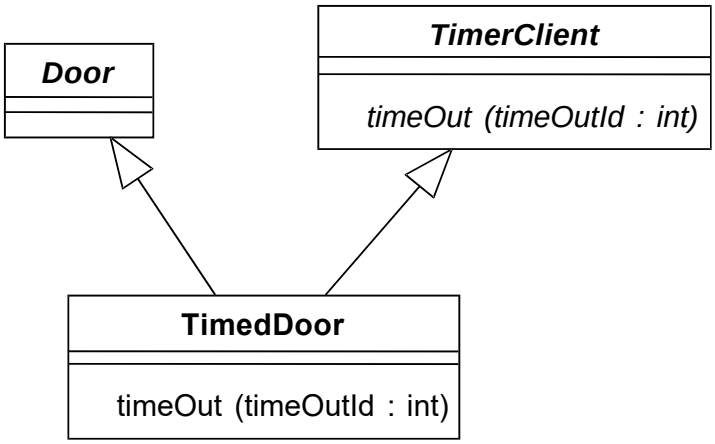
Solutions

Think about the fact that clients of an object do not need to access it through the interface of the object itself.

Think they can access it through delegation, or through a base class of the object (i.e., polymorphycally).

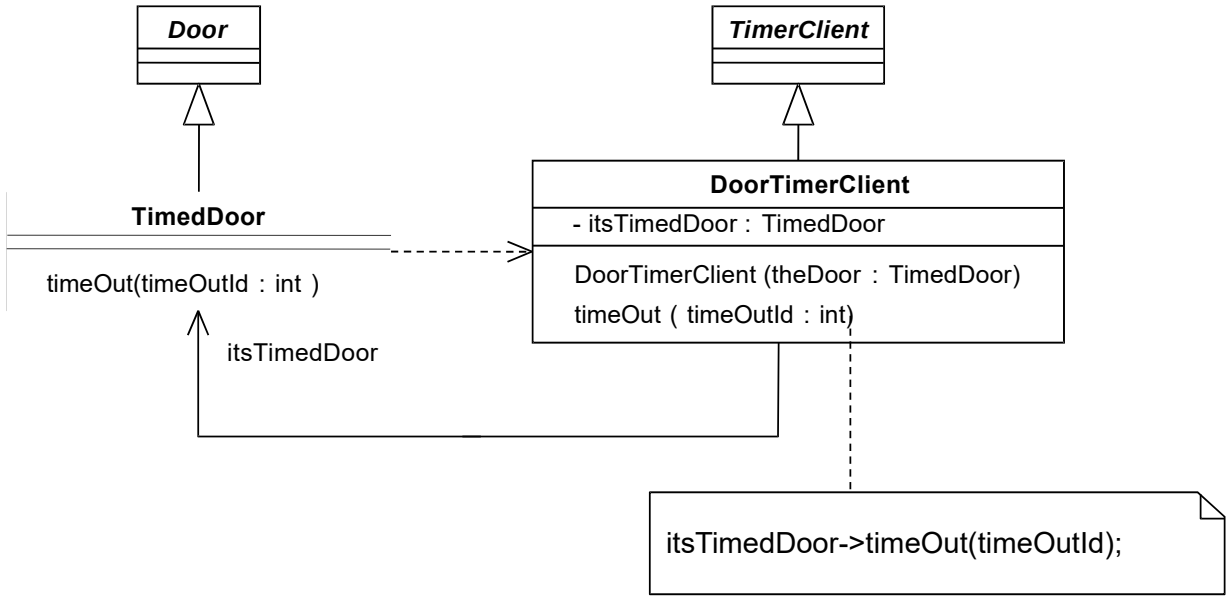


Separation through Multiple Inheritance

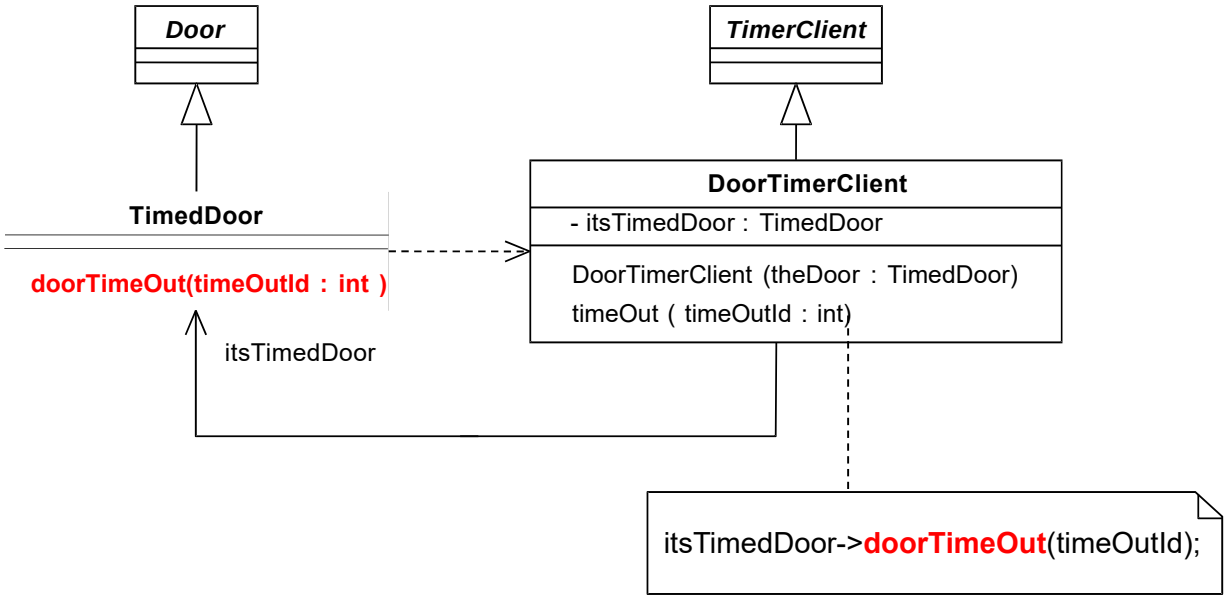


Neither clients of Door and TimerClient do not depend on each other.

Separation through Delegation



Prevents the coupling of Door clients to TimerClient.



TimedDoor can have a different interface from TimedClient interface.